Architecture des ordinateurs
deuxième partie des annales

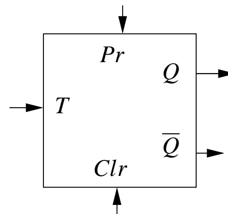
Arnaud Giersch, Benoît Meister et Frédéric Vivien

1 TD 6 : Circuits séquentiels

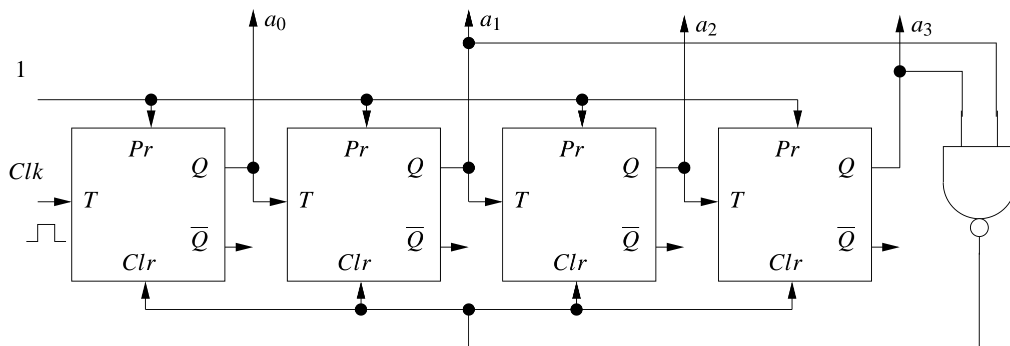
1. **Bascules** *T*
   On considère une bascule dont la table de vérité est la suivante. On considère que ε est petit par rapport à un cycle d'horloge.

Pr	Clr	T	$Q_{t-\varepsilon}$	$\overline{Q_{t-\varepsilon}}$	Q_t	$\overline{Q_t}$
0	0	x	y	$\overline{y}$	#	#
0	1	x	y	$\overline{y}$	1	0
1	0	x	y	$\overline{y}$	0	1
1	1	0	y	$\overline{y}$	$\overline{y}$	y
1	1	1	y	$\overline{y}$	y	$\overline{y}$

   Dans cette table, les entrées Pr et Clr signifient respectivement `Preset` et `Clear` ; elles sont actives sur niveau bas. L'entrée T reçoit un signal d'horloge et est active sur niveau bas. Les variables x et y prennent indifféremment les valeurs 0 ou 1. Le symbole # signifie qu'il est impossible de déterminer la valeur logique de la variable auquel il se réfère. La présence du symbole # correspond au cas où les deux signaux Pr et Clr sont simultanément actifs, ce qui est interdit.
   Ce type de bascule est appelée **bascule T** (« *T* » pour *time*). À l'instar d'une bascule D, l'activation d'une bascule T peut se faire sur les fronts montants ou descendants du signal d'entrée. Cela permet qu'il n'y ait qu'un changement d'état par cycle d'horloge (Q_t dépend de $Q_{t-\varepsilon}$ et non de Q_{t-1}). Ici, l'activation de notre bascule T se fera sur front descendant. Une telle bascule est représentée de la manière suivante.

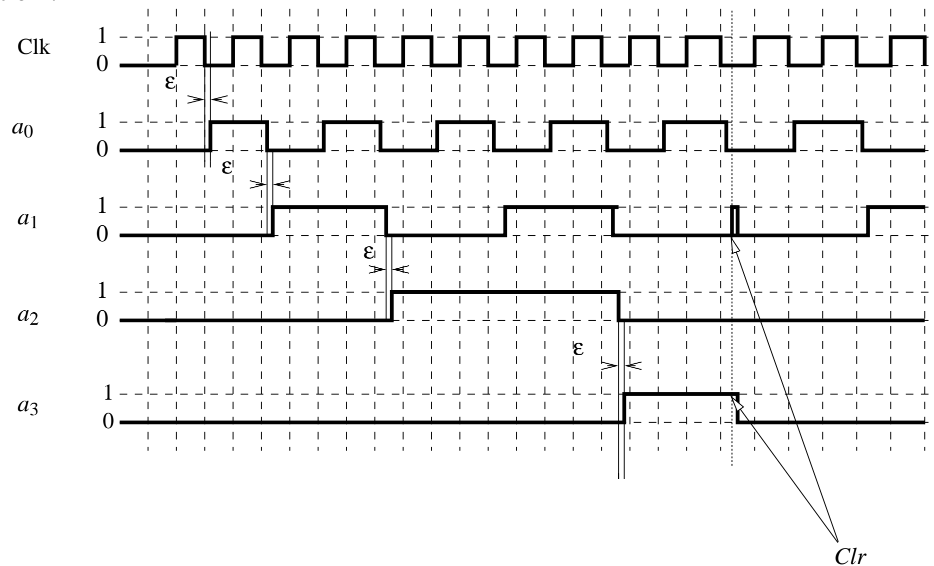


   On considère à présent le dispositif constitué de quatre bascules T montées en cascade selon le schéma ci-dessous.



(a) Compléter le chronogramme suivant en supposant que le temps de traversée d'une bascule T est ε et que le temps de traversée d'une porte logique NAND est négligeable (par rapport à ε).

Correction :



Quelques explications relatives à ce chronogramme :

- *Au départ, toutes les sorties a_i sont au niveau 0. Comme les sorties a_1 et a_3 sont à 0, l'entrée Clear de toutes les bascules T est à 1 (inactive). De plus, notons que l'entrée Preset de chaque bascule T est toujours fixée à 1.*
- *Au premier passage de l'horloge au niveau bas, la sortie a_0 est inversée (l'inversion se faisant sur les front descendants). On a donc : $a_0 = 1$.*
- *Au second front descendant, a_0 repasse à 0, ce qui constitue un front descendant à l'entrée de la deuxième bascule. Sa sortie, a_1, passe donc au niveau 1.*
- *Une bascule change de niveau à chaque fois que son entrée reçoit un front descendant. La sortie a_0 produira donc un front descendant au bout de deux fronts descendants reçus en entrée. De même, la sortie a_1 changera de niveau à chaque front descendant produit par a_0, et produira un front descendant au bout de deux fronts descendants produits par a_0. Ainsi de suite pour a_2, et a_3. Finalement, a_0 change de niveau après 2 changements de niveau de l'horloge, a_1 change de niveau après 2 changements de niveau de a_0 (donc après 4 changements de niveau de l'horloge), et ainsi de suite (a_3 : 8 changements de niveau d'horloge, a_4 : 16 changements).*

(b) Quelle est la signification de la représentation décimale du nombre binaire $a_3a_2a_1a_0$? Quelle est la fonction du dispositif ?

Correction : *La sortie $a_3a_2a_1a_0$ compte le nombre de cycles d'horloge sur 4 bits. Mais la porte nand met les signaux Clear de toutes les bascules à 0 (actifs) lorsque a_1 et a_3 sont égaux à 1. L'activation de tous les Clear met toutes les sorties a_i à 0. Ce cas arrive pour la première fois lorsque $a_3a_2a_1a_0 = 10$. Comme $a_3a_2a_1a_0 = 10$ est alors remis à 0, ceci n'arrive* **que** *lorsque $a_3a_2a_1a_0 = 10$. La sortie $a_3a_2a_1a_0$ compte donc le nombre de cycles d'horloge reçus en entrée, modulo 10.*

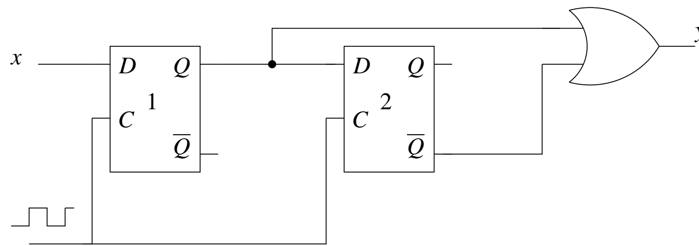
2. **Feux de circulation**

On veut faire un circuit gérant les feux de circulation d'un croisement entre deux routes, de directions Nord/Sud et Est/Ouest. Les feux, qui sont soit rouges (signal de valeur 0) soit verts (signal de valeur 1), passent alternativement d'une couleur à l'autre. Lorsqu'un piéton souhaite traverser le croisement, il appuie sur un bouton pour faire passer *tous* les feux au rouge.

(a) Modéliser le bouton pour les piétons à l'aide de deux bascules D : lorsque le piéton appuie sur le bouton, cela fait passer son entrée x de la valeur 1 à la valeur 0 pendant un temps supérieur à un cycle d'horloge. x revient ensuite à la valeur 1. A la pression du bouton, sa sortie y doit produire un signal à 0 pendant un

cycle d'horloge puis revenir à 1 (sa valeur normale). On supposera dans tout l'exercice que le temps de passage des portes logiques est négligeable devant la durée d'un cycle, et que les bascules se déclenchent sur front descendant.

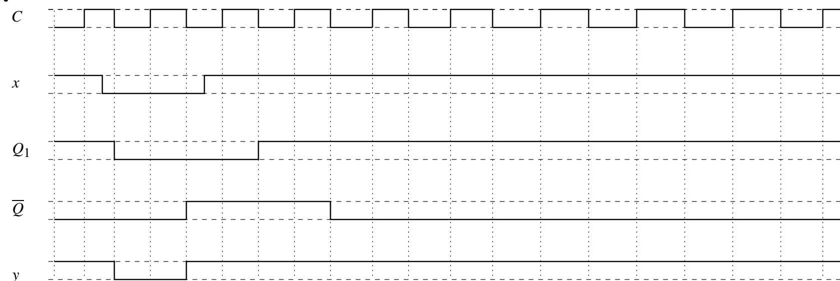
Correction :



Ce circuit se comporte comme un détecteur de front descendant (avec un not *en sortie) : Au départ,* $x = 1$*, et* $\overline{Q} = 0$*, donc y est à 1. Lorsque x passe à 0,* $y = 0$*. Mais* $\overline{Q}$ *passe à 1 au cycle suivant, et donc y repasse à 1. Lorsque x repasse à 1, y reste à 1.*

Montrer la validité du circuit à l'aide d'un chronogramme.

Correction :

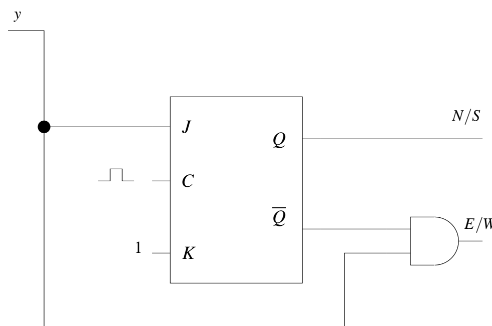


En pratique, la période de l'horloge associée au bouton est très petite (typiquement 20ms) devant le temps de passage du piéton (de l'ordre de la minute). Pour que la durée du signal corresponde à ce temps, un diviseur de fréquence est placé entre la sortie du bouton et l'entrée du reste du circuit. La période d'horloge considérée dans la suite est égale à ce temps, elle est donc elle aussi très grande devant celle de l'horloge associée au bouton.

(b) On veut à présent modéliser le circuit mettant tous les feux au rouge pendant un cycle, à la réception du signal donné par l'interrupteur. À l'aide d'une bascule JK dont l'horloge est synchronisée avec celle du bouton piéton, construire 3 circuits différents, distingués par le feu que l'on met au vert après le passage du piéton :

i. Celui de la direction Nord/Sud.

Correction :

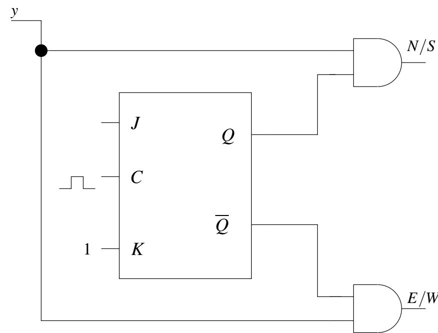


– *Fonctionnement normal :* $y = 1 \Rightarrow J = K = 1$ *: à chaque cycle, la valeur de Q (et de* $\overline{Q}$*) est inversée. Les feux passent donc alternativement au vert puis au rouge.*
– *Lorsque le bouton est pressé, J passe à la valeur 0, ainsi qu'une entrée de la porte* and *A. A ce cycle d'horloge, Q prend la valeur 0. La porte* and *A place la valeur de sortie* E/W *à 0.*

– *Au cycle suivant, on a à nouveau $J = K = 1$: Q prend la valeur 1 ⇒ l'axe N/S passe au vert.*

ii. Celui qui était vert avant qu'on appuie sur le bouton.

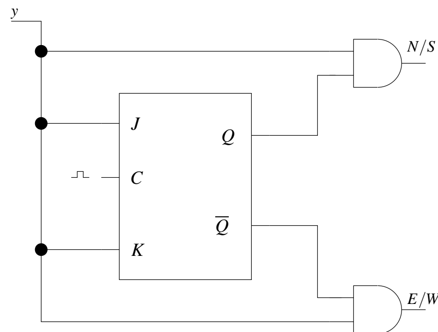
Correction :



Ici, les deux portes and *mettent la valeur des sorties N/S et E/W à 0 pendant un cycle. Toutefois, les sorties de la bascule JK continuent de changer de valeur une fois pendant ce cycle. Au cycle suivant, la valeur des sorties N/S et E/W est la même qu'avant le passage du piéton.*

iii. Celui qui était rouge avant qu'on appuie sur le bouton.

Correction :



Ici, les entrées J et K sont reliées à l'entrée y. Pendant que l'on annule les sorties N/S et E/W, les entrées J et K sont elles aussi mises á la valeur 0. Les valeurs de sortie Q et $\overline{Q}$ ne sont donc pas modifiées pendant ce temps, mais seulement au cycle suivant (lorsque J et K sont remises à 1). Les valeurs prises par les sorties N/S et E/W au cycle suivant sont donc l'inverse de ce qu'elles étaient avant le passage du piéton.

2 TD 7 : Circuits séquentiels (suite)

Multiplicateur de mots de 3 bits

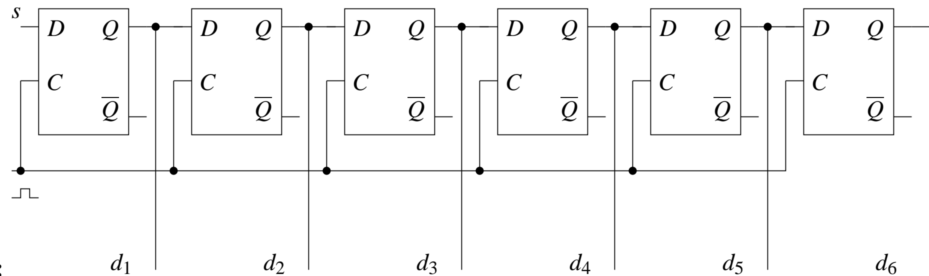
On rappelle qu'un registre à décalages sur n bits est pourvu de n sorties d_1 à d_n et d'une entrée s (dite « entrée série »). Au temps $t+1$, la valeur de chaque sortie $d_i, i \in [2..n]$, est égale à la valeur prise par la sortie d_{i-1} au temps t. La valeur de d_1 au temps $t+1$ est égale à la valeur de l'entrée s au temps t.

1. Rappeler le fonctionnement d'une bascule D simple.

   Correction : *cf. le cours*

2. Réaliser un registre à décalages sur 6 bits à l'aide de bascules D.

Correction :



3. Expliciter les valeurs prises par les sorties d_1 à d_6 avec comme entrée le mot 110. Écrire l'évolution des valeurs de sortie pour les temps $t = 0$ à 6. La valeur d'entrée avant et après le mot est de 0.

   Correction :

t	d_1	d_2	d_3	d_4	d_5	d_6
0	0	0	0	0	0	0
1	1	0	0	0	0	0
2	1	1	0	0	0	0
3	0	1	1	0	0	0
4	0	0	1	1	0	0
5	0	0	0	1	1	0
6	0	0	0	0	1	1

4. Détailler la multiplication de deux nombres de 3 bits, par exemple $A = 110_b$ et $B = 101_b$, en une suite d'additions.

   Correction :

$$\begin{array}{rrr} & & 110 \\ \times & & 101 \\ \hline & 1 \times 110 = & 110 \\ + & 0 \times 1100 = & 0 \\ + & 1 \times 11000 = & 11000 \\ \hline & & 11110 \end{array}$$

   Expliquer où intervient un décalage lors de l'exécution de cette opération.

   Correction : *La multiplication de* 110_b *par* 101_b *se décompose ainsi :*

$$110_b \times 101_b = 110 \times 1.2^0 + 110 \times 0.2^1 + 110 \times 1.2^2$$

   La multiplication d'un nombre binaire par 2 équivaut au décalage d'un cran à gauche de ce nombre binaire.

5. On dispose d'un additionneur sur 6 bits, prenant en entrée deux entiers sur 6 bits $C = c_6c_5c_4c_3c_2c_1$ et $F = f_6f_5f_4f_3f_2f_1$, et calculant en sortie la somme $C + F = S = s_6s_5s_4s_3s_2s_1$. Fabriquer un multiplicateur d'entiers sur 3 bits (avec résultat sur 6 bits) à l'aide d'un registre à décalages sur 6 bits, de l'additionneur 6 bits et d'éventuelles portes logiques combinatoires et/ou séquentielles. On considère que le temps de passage des portes logiques combinatoires et celui de l'additionneur sont négligeables devant la période de l'horloge.

   Correction : *On utilise un registre à décalage pour effectuer les décalages à gauche (bien qu'ils aient l'air à droite) sur A. Il reste à multiplier par 1 ou 0 les nombres décalés (selon la valeur du bit de B correspondant), et à les additionner entre eux. Au préalable, il faut* charger *le nombre A dans le registre à décalage, ce qui prend 3 cycles d'horloge. Pour synchroniser correctement le décalage avec la multiplication par les bits de B, on peut « retarder » de 3 cycles la prise en compte des bits de B, par exemple à l'aide de 3 portes D. Les bits de A sont entrés du bit de poids le plus fort au bit de poids le plus faible, alors que les bits de B sont entrés dans l'ordre inverse, c'est-à-dire du bit de poids le plus faible au bit de poids le plus fort. L'utilisation d'un additionneur 6 bits permet de ne pas avoir de débordement (ou* overflow*). Dans le schéma de la figure 1, toutes les bascules D sont reliées à la même horloge.*

6. Donner le nombre de cycles nécessaires à l'exécution d'une multiplication.

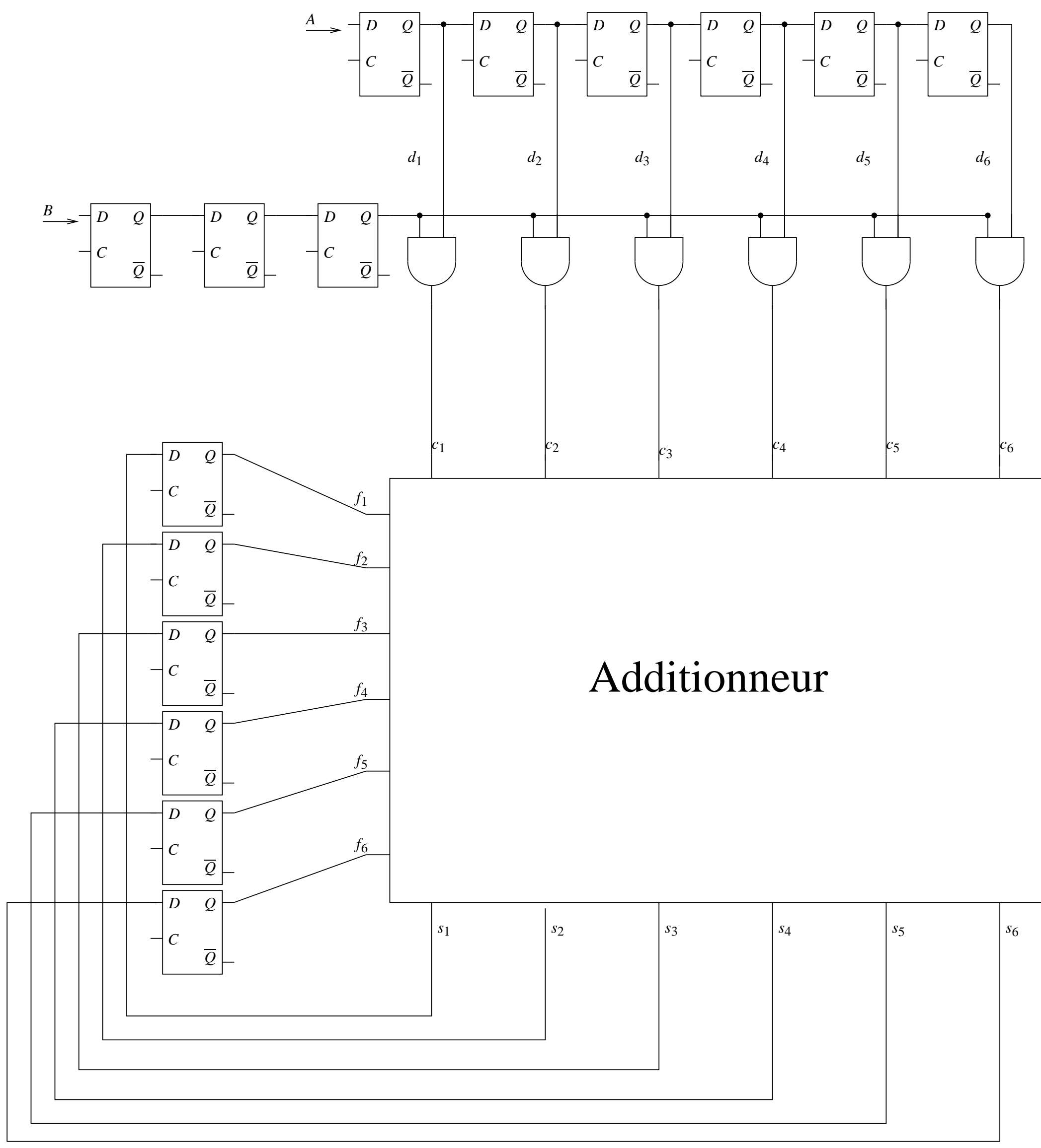


FIG. 1 – Multiplicateur réalisé au moyen d'un additionneur et d'un registre à décalage.

Correction : *Les 3 opérandes de l'addition sont produits aux temps 3, 4 et 5 à l'entrée C de l'additionneur. Cette valeur est répercutée à l'entrée F au cycle suivant. L'addition des 3 opérandes se termine au temps 5. Ici, le temps d'exécution de la multiplication est de 5 cycles d'horloge.*

7. Rappeler le fonctionnement d'une bascule D pourvue d'entrées *Clear* et *Preset* actives au niveau bas.

   Correction : *On peut forcer la valeur prise par les sorties d'une bascule D par l'utilisation des entrées* Clear, *qui place la valeur de Q à 0, et* Preset, *qui place la valeur de Q à 1. Ces valeurs de sortie sont prises quelque soit la valeur à l'entrée D. Dans le cas où elles sont « actives au niveau bas », ces entrées font leur effet lorsque leur valeur est mise à 0.*

8. Montrer comment on peut réduire le temps d'exécution de la multiplication si l'on utilise ce type de bascule pour la fabrication du registre à décalages.

   Correction : *Le chargement de la donnée A dans le registre à décalages peut être fait en 1 cycle par les entrées* Preset *des 3 premières bascules, comme le montre le circuit de la figure 2.*

9. Quel est le temps d'exécution de la multiplication pour ce nouveau circuit ?

   Correction : *Le temps d'exécution de la multiplication par ce nouveau circuit est de 3 cycles d'horloge.*

3 Examen de contrôle continu

3.1 Arithmétique des ordinateurs

Dans toute cette section les nombres seront **codés sur 8 bits**.

3.1.1 Codage des entiers

1. Codez en binaire pur les nombres décimaux : 15, 122 et 223.
   (a) $15 = 1+2+4+8$. *D'où* $15_{10} = 00001111_2$.
   (b) $122 = 64+32+16+8+2$. *D'où* $122_{10} = 01111010_2$.
   (c) $223 = 128+64+16+8+4+2+1$. *D'où* $223_{10} = 11011111_2$.
2. Codez, en représentation avec signe et valeur absolue, les nombres décimaux -122, -15, 15 et 122.
   (a) $-122_{10} = 11111010_2$.
   (b) $-15_{10} = 10001111_2$.
   (c) $15_{10} = 00001111_2$.
   (d) $122_{10} = 01111010_2$.
3. Codez, dans un système avec complément à deux, les nombres décimaux -122, -15, 15 et 122.
   (a) -122_{10}. *Dans un tel système, les nombres négatifs sont représentés par le complément à deux de leur valeur absolue. Codage en binaire pur de* 122 : 01111010 *; codage en complément à un :* $C_1(122) = 10000101$ *; codage en complément à deux :* $C_2(122) = 10000110$. *D'où :* $-122_{10} = 10000110_2$.
   (b) -15_{10}. *Codage en binaire pur de* 15 : 00001111 *; codage en complément à un :* $C_1(15) = 11110000$ *; codage en complément à deux :* $C_2(15) = 11110001$. *D'où :* $-15_{10} = 11110001_2$.
   (c) 15. *Dans un tel système, les nombres positifs sont représentés par leur codage binaire pur :* $15_{10} = 00001111_2$.
   (d) 122. $122_{10} = 01111010_2$.

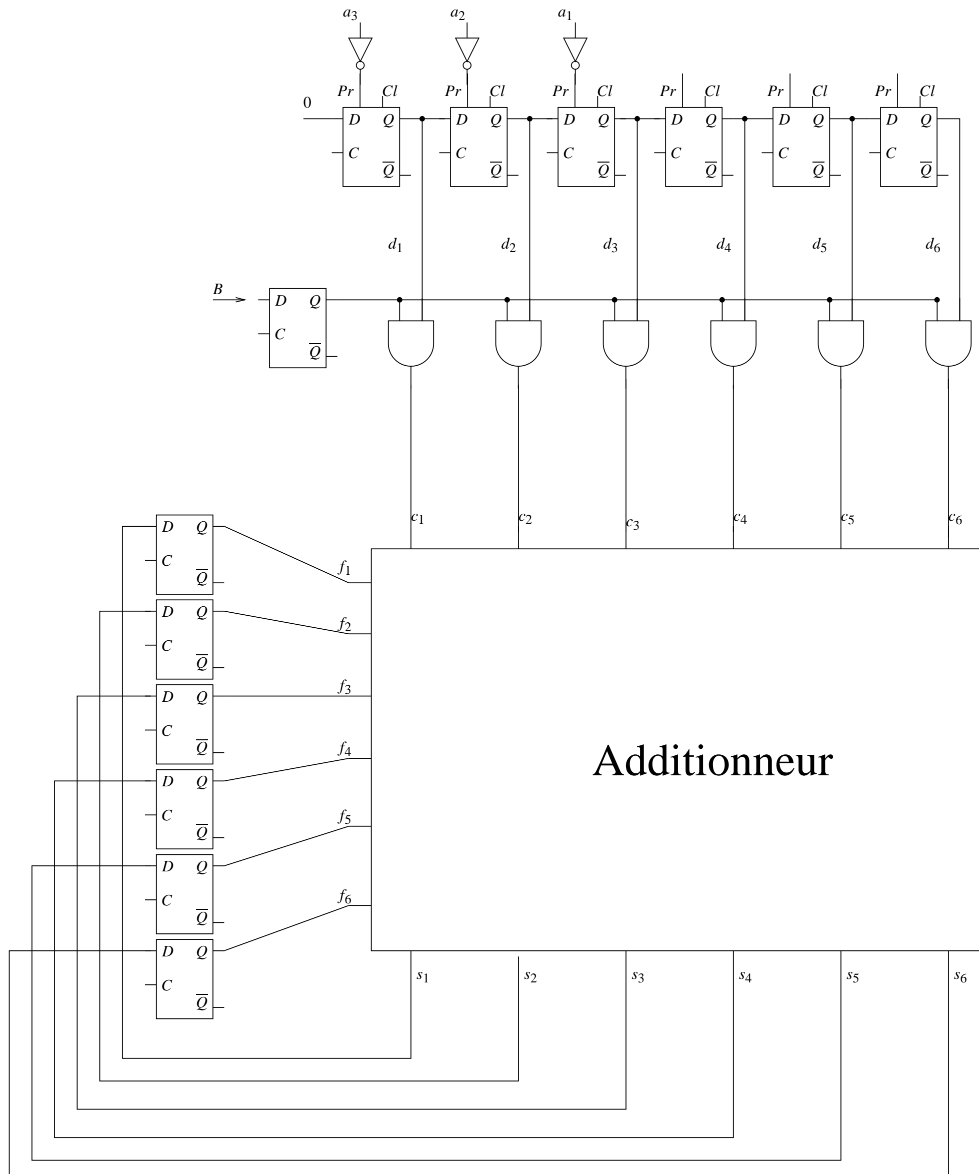


FIG. 2 – Deuxième multiplicateur réalisé au moyen d'un additionneur et d'un registre à décalage.

3.1.2 Addition d'entiers

1. Calculez l'addition des deux nombres décimaux -122 et 15 dans un système avec complément à deux.

   Dans un tel système, on effectue l'addition des représentations des deux nombres : le codage binaire pur du nombre positif et le codage en représentation à deux du nombre négatif.

$$\begin{array}{rr} & 10000110 \\ + & 00001111 \\ \hline & 10010101 \end{array}$$

2. Convertissez le résultat en base 10.

   Le bit de poids fort du nombre binaire 10010101 *étant « 1 », le nombre est négatif, et ce code est celui du complément à deux de la valeur absolue du nombre. Pour trouver la valeur absolue, on prend le complément à deux de* 10010101 *(le complément à deux est une fonction involutive : le complément à deux du complément à deux est l'identité).*

   $C_1(10010101) = 01101010$ *et* $C_2(10010101) = 01101011$. *Or* $01101011_2 = 0\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0 = 64+32+8+2+1 = 107$. *Donc :* $10010101_2 = -107_{10}$.

3.2 Algèbre de Boole

1. Proposez une expression booléenne ayant pour table de vérité la table ci-dessous :

A	B	C	D	$f(A,B,C,D)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

   La solution classique sous forme de somme de produits :

$$\begin{aligned} f(A,B,C,D) = \quad & \overline{A}.\overline{B}.\overline{C}.\overline{D}+\overline{A}.\overline{B}.\overline{C}.D+\overline{A}.\overline{B}.C.\overline{D}+\overline{A}.\overline{B}.C.D+\overline{A}.B.\overline{C}.D \\ & +\overline{A}.B.C.D+A.\overline{B}.\overline{C}.\overline{D}+A.\overline{B}.\overline{C}.D+A.\overline{B}.C.\overline{D}+A.\overline{B}.C.D+A.B.C.D \end{aligned}$$

2. Simplifiez l'expression booléenne de la question précédente au moyen d'une table de Karnaugh.

   La figure 3 présente la table de Karnaugh désirée.

   On en déduit l'expression booléenne simplifiée :

$$f(A,B,C,D) = \overline{B}+C.D+\overline{A}.D.$$

$CD \backslash AB$	AB	$\overline{A}B$	$\overline{A}\overline{B}$	$A\overline{B}$
CD	1	1	1	1
$\overline{C}D$	0	1	1	1
$\overline{C}\overline{D}$	0	0	1	1
$C\overline{D}$	0	0	1	1

FIG. 3 – Table de Karnaugh correspondant à la table de vérité et à l'expression précédentes.

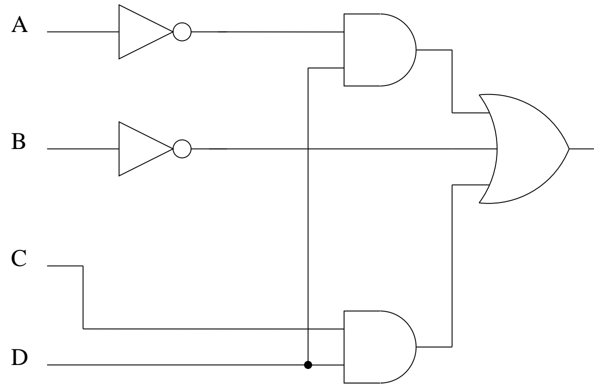


FIG. 4 – Circuit logique de l'expression simplifiée par table de Karnaugh.

3.3 Circuits logiques

1. Réalisez le circuit logique correspondant à l'expression booléenne simplifiée trouvée à la question précédente.
   Le circuit est présenté figure 4.
2. Complétez le chronogramme du circuit de la figure 5 en considérant que toutes les portes logiques du circuit ont le même temps de passage Δt.

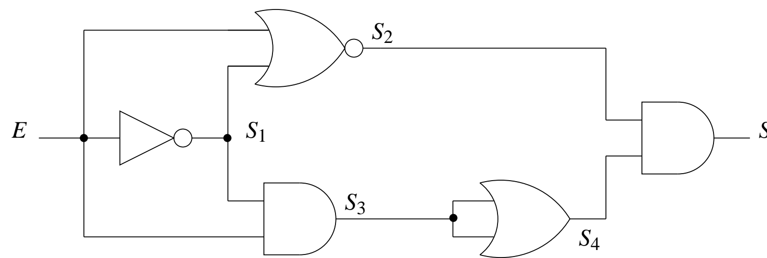


FIG. 5 – Circuit à étudier.

   Le chronogramme complété est présenté figure 6.

3. Quelle est la fonction du circuit de la figure 5 ?
   Ce circuit est un détecteur d'impulsion.

4 TP 1 : Assembleur SPARC

Remarques préliminaires :

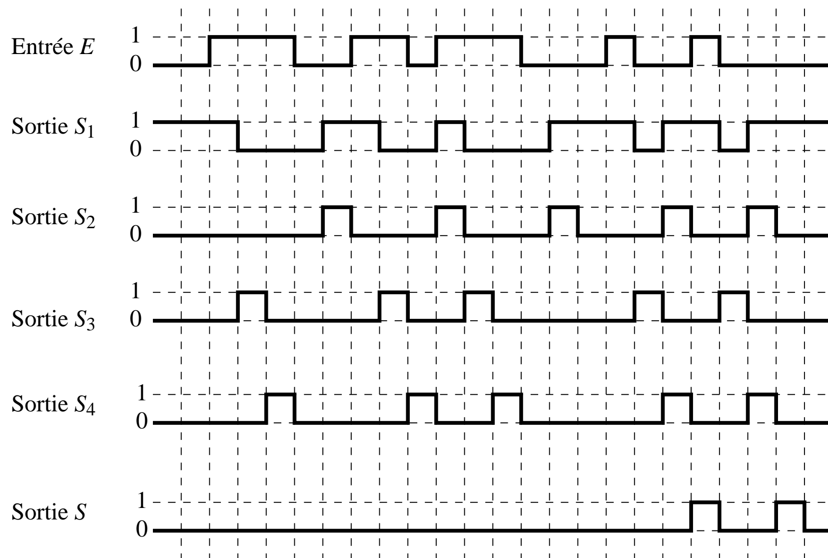


FIG. 6 – Chronogramme complété.

– le nom d'un fichier contenant un programme assembleur doit obligatoirement avoir le suffixe `.s` ;
– on passe d'un programme assembleur à un fichier exécutable en utilisant `gcc` de la même manière qu'avec un programme C.

1. Étude d'un programme en assembleur SPARC

   (a) Récupérez le programme `addition.s` qui réalise la somme de 2 entiers. Compilez en utilisant la commande `gcc -o addition addition.s` et exécutez-le.

   URL : `http://icps.u-strasbg.fr/~vivien/Enseignement/Archi-2001-2002/addition.s`.

   Étudiez ce programme et rajoutez des commentaires explicatifs dans ce fichier addition.s.

   Correction :

```
! Fonction addition
!       -> retourne la somme de ses deux arguments
!-----------------------------------------------------------

.section        ".text"                 ! -> code
        .align 4                        !    aligné sur 4 octets
        .global add                     ! exporte le symbole « add »

add:
        save %sp,-64,%sp                ! réserve de l'espace sur la pile
        add %i0,%i1,%i0                 ! calcule la somme des paramètres,
                                        ! le résultat est la valeur de retour
                                        ! de la fonction
        ret                             ! retour de la fonction add
        restore


! Programme principal
!-----------------------------------------------------------

.section        ".data"                 ! -> données
        .align 8                        !    alignées sur 8 octets
```

```
.PRINTF1:
        .asciz  "Entrez a et b : "      ! chaîne de caractères avec zéro
                                        ! terminal
.SCANF:
        .asciz  "%d %d"                 ! idem
.PRINTF2:
        .asciz  "c : %d\n"              ! idem

.section        ".text"                 ! -> code
        .align 4                        !    aligné sur 4 octets
        .global main                    ! exporte le symbole « main »

main:
        save %sp,-104,%sp               ! réserve de l'espace sur la pile:
                                        ! 96 + 8 octets pour deux variables
                                        ! locale aux adresses %fp-4 et %fp-8

        ! printf
        sethi %hi(.PRINTF1),%o0         ! %o0 <- .PRINTF1
        or %o0,%lo(.PRINTF1),%o0
        call printf                     ! appel de « printf (.PRINTF1) »
        nop                             ! « nop » après un « call »

        ! scanf
        add %fp,-4,%o1                  ! %o1 <- %fp-4
        add %fp,-8,%o2                  ! %o2 <- %fp-8
        sethi %hi(.SCANF),%o0           ! %o0 <- .SCANF
        or %o0,%lo(.SCANF),%o0
        call scanf                      ! appel de
                                        ! « scanf (.SCANF1, %fp-4, %fp-8) »
        nop                             ! « nop » après un « call »

        ! addition
        ld [%fp-4],%o0                  ! %o0 <- [%fp-4]
        ld [%fp-8],%o1                  ! %o1 <- [%fp-8]
        call add                        ! appel de « add ([%fp-4], [%fp-8]) »
                                        ! résultat dans %o0
        nop                             ! « nop » après un « call »

        ! printf
        mov %o0,%o1                     ! %o1 <- %o0
        sethi %hi(.PRINTF2),%o0         ! %o0 <- .PRINTF2
        or %o0,%lo(.PRINTF2),%o0
        call printf                     ! appel de « printf (.PRINTF2, %o0) »
        nop                             ! « nop » après un « call »

        ret                             ! retour de la fonction  main
        restore
```

(b) Réalisez un programme C faisant appel à une fonction prenant 8 paramètres, produisez le programme assembleur correspondant (option `-S` de `gcc` : `gcc -S fichiersource`).

Déterminez ensuite quels paramètres sont placés dans les registres, lesquels ne le sont pas et trouvez l'emplacement mémoire de ces derniers.

Correction :

– *le programme C :*

```
#include <stdio.h>
```

```
int add8 (int a, int b, int c, int d, int e, int f, int g, int h)
{
    return a + b + c + d + e + f + g + h;
}

int main (void)
{
    printf ("somme 1..8 = %d\n",
            add8 (1, 2, 3, 4, 5, 6, 7, 8));

    return 0;
}
```

– *le code assembleur correspondant :*

```
        .file   "8arg.c"
gcc2_compiled.:
.section        ".text"
        .align 4
        .global add8
        .type    add8,#function
        .proc   04
add8:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        st      %i0, [%fp+68]
        st      %i1, [%fp+72]
        st      %i2, [%fp+76]
        st      %i3, [%fp+80]
        st      %i4, [%fp+84]
        st      %i5, [%fp+88]
        ld      [%fp+68], %o0
        ld      [%fp+72], %o1
        add     %o0, %o1, %o0
        ld      [%fp+76], %o1
        add     %o0, %o1, %o0
        ld      [%fp+80], %o1
        add     %o0, %o1, %o0
        ld      [%fp+84], %o1
        add     %o0, %o1, %o0
        ld      [%fp+88], %o1
        add     %o0, %o1, %o0
        ld      [%fp+92], %o1
        add     %o0, %o1, %o0
        ld      [%fp+96], %o1
        add     %o0, %o1, %o0
        mov     %o0, %i0
        b       .LL2
         nop
.LL2:
        ret
        restore
.LLfe1:
        .size    add8,.LLfe1-add8
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "somme 1..8 = %d\n"
.section        ".text"
```

```
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save    %sp, -120, %sp
        !#PROLOGUE# 1
        mov     7, %o0
        st      %o0, [%sp+92]
        mov     8, %o0
        st      %o0, [%sp+96]
        mov     1, %o0
        mov     2, %o1
        mov     3, %o2
        mov     4, %o3
        mov     5, %o4
        mov     6, %o5
        call    add8, 0
         nop
        mov     %o0, %o1
        sethi   %hi(.LLC0), %o2
        or      %o2, %lo(.LLC0), %o0
        call    printf, 0
         nop
        mov     0, %i0
        b       .LL3
         nop
.LL3:
        ret
        restore
.LLfe2:
        .size    main,.LLfe2-main
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

On remarque que les 6 premiers paramètres sont passés par les registres `%o0` à `%o5` (`%i0` à `%i5` dans la fonction), les deux derniers sont passés sur la pile aux adresses `%sp+92` et `%sp+96` (`%fp+92` et `%fp+96` dans la fonction).

2. Exercices de programmation

   (a) Écrivez un programme assembleur calculant la factorielle d'un entier de manière itérative (une seule fonction principale contenant une boucle).

   Correction :

```
!!!
!!! calcul de la factorielle d'un entier, version itérative
!!!

! Programme principal
!------------------------------------------------------------

.section        ".data"                 ! -> données
        .align 8
.PRINTF1:
        .asciz  "n? "
.SCANF:
        .asciz  "%u"
.PRINTF2:
```

```
        .asciz "n! = %u\n"


.section        ".text"                 ! -> code
        .align 4
        .global main

main:
        save %sp, -96, %sp              ! réserve de la place sur la pile
                                        ! pour un entier [%fp-4] (mais on
                                        ! arrondit à un multiple de 8)

        sethi %hi(.PRINTF1), %o0
        or %o0, %lo(.PRINTF1), %o0
        call printf                     ! printf (.PRINTF1)
        nop

        sethi %hi(.SCANF), %o0
        or %o0, %lo(.SCANF), %o0
        add %fp, -4, %o1
        call scanf                      ! scanf (.SCANF, %fp-4)
        nop

                                        ! -> calcul de [%fp-4]! dans %l1,
                                        !   utilisation de %l0 comme compteur

        ld [%fp-4], %l0                 ! %l0 <- [%fp-4]
        mov 1, %l1                      ! %l1 <- 1
loop:
        cmp %l0, 1                      ! while (%l0 > 1) {
        ble end_loop                    !
        nop                             !
        umul %l0, %l1, %l1              !     %l1 <- %l1 * %l0
        dec %l0                         !     %l0 --
        b loop                          ! }
        nop

end_loop:
        sethi %hi(.PRINTF2), %o0
        or %o0, %lo(.PRINTF2), %o0
        mov %l1, %o1
        call printf                     ! printf (.PRINTF2, %l1)
        nop

        clr %i0                         ! return 0
        ret
        restore
```

(b) Écrivez un programme assembleur calculant la factorielle d'un entier de manière *récursive*.

Correction :

```
!!!
!!! calcul de la factorielle d'un entier, version récursive
!!!

! fonction fact
!       -> retourne la factorielle de son argument
!------------------------------------------------------------
```

```
.section        ".text"                 ! -> code
        .align 4
        .global fact

fact:
        save %sp, -96, %sp

        cmp %i0, 1                      ! if (%i0 > 1) {
        ble end_fact1                   !
        nop                             !
        sub %i0, 1, %o0                 !     %o0 <- %i0 - 1
        call fact                       !     %o0 <- fact (%o0)
        nop                             !
        umul %i0, %o0, %i0              !     %i0 <- %o0 * %i0
        b end_fact                      ! } else {
        nop                             !
end_fact1:                              !
        mov 1, %i0                      !     %i0 <- 1
                                        ! }
end_fact:
        ret                             ! return %i0
        restore


! Programme principal
!-------------------------------------------------------------

.section        ".data"                 ! -> données
        .align 8
.PRINTF1:
        .asciz  "n? "
.SCANF:
        .asciz  "%u"
.PRINTF2:
        .asciz "n! = %u\n"

.section        ".text"                 ! -> code
        .align 4
        .global main

main:
        save %sp, -96, %sp              ! réserve de la place sur la pile
                                        ! pour un entier [%fp-4] (mais on
                                        ! arrondit à un multiple de 8)

        sethi %hi(.PRINTF1), %o0
        or %o0, %lo(.PRINTF1), %o0
        call printf                     ! printf (.PRINTF1)
        nop

        sethi %hi(.SCANF), %o0
        or %o0, %lo(.SCANF), %o0
        add %fp, -4, %o1
        call scanf                      ! scanf (.SCANF, %fp-4)
        nop

        ld [%fp-4], %o0
        call fact                       ! %o0 <- fact ([%fp-4])
```

```
        nop

        mov %o0, %o1
        sethi %hi(.PRINTF2), %o0
        or %o0, %lo(.PRINTF2), %o0
        call printf                     ! printf (.PRINTF2, %o0)
        nop

        clr %i0                         ! return 0
        ret
        restore
```

(c) Modifiez-le programme précédent pour qu'il affiche à chaque étape de la récursion les valeurs des pointeurs de pile (`%sp` et `%fp`).

Correction :

```
!!!
!!! calcul de la factorielle d'un entier, version récursive,
!!! affichage de %fp et %sp
!!!

! fonction fact
!       -> retourne la factorielle de son argument
!------------------------------------------------------------

.section        ".data"                 ! -> données
        .align 8
.PRINTF_DEBUG:
        .asciz "# fact (%u): %%fp = %p, %%sp = %p\n"

.section        ".text"                 ! -> code
        .align 4
        .global fact

fact:
        save %sp, -96, %sp

        sethi %hi(.PRINTF_DEBUG), %o0
        or %o0, %lo(.PRINTF_DEBUG), %o0
        mov %i0, %o1
        mov %fp, %o2
        mov %sp, %o3
        call printf                     ! printf (.PRINTF_DEBUG,
                                        !         %i0, %fp, %sp)
        nop

        cmp %i0, 1                      ! if (%i0 > 1) {
        ble end_fact1                   !
        nop                             !
        sub %i0, 1, %o0                 !     %o0 <- %i0 - 1
        call fact                       !     %o0 <- fact (%o0)
        nop                             !
        umul %i0, %o0, %i0              !     %i0 <- %o0 * %i0
        b end_fact                      ! } else {
        nop                             !
end_fact1:                              !
        mov 1, %i0                      !     %i0 <- 1
                                        ! }
end_fact:
```

```
        ret                             ! return %i0
        restore


! Programme principal
!------------------------------------------------------------

.section        ".data"                 ! -> données
        .align 8
.PRINTF1:
        .asciz  "n? "
.SCANF:
        .asciz  "%u"
.PRINTF2:
        .asciz "n! = %u\n"

.section        ".text"                 ! -> code
        .align 4
        .global main

main:
        save %sp, -96, %sp              ! réserve de la place sur la pile
                                        ! pour un entier [%fp-4] (mais on
                                        ! arrondit à un multiple de 8)

        sethi %hi(.PRINTF1), %o0
        or %o0, %lo(.PRINTF1), %o0
        call printf                     ! printf (.PRINTF1)
        nop

        sethi %hi(.SCANF), %o0
        or %o0, %lo(.SCANF), %o0
        add %fp, -4, %o1
        call scanf                      ! scanf (.SCANF, %fp-4)
        nop

        ld [%fp-4], %o0
        call fact                       ! %o0 <- fact ([%fp-4])
        nop

        mov %o0, %o1
        sethi %hi(.PRINTF2), %o0
        or %o0, %lo(.PRINTF2), %o0
        call printf                     ! printf (.PRINTF2, %o0)
        nop

        clr %i0                         ! return 0
        ret
        restore
```

Exemple d'exécution :

```
$ ./fact_r_print
n? 6
# fact (6): %fp = ffbef800, %sp = ffbef7a0
# fact (5): %fp = ffbef7a0, %sp = ffbef740
# fact (4): %fp = ffbef740, %sp = ffbef6e0
# fact (3): %fp = ffbef6e0, %sp = ffbef680
# fact (2): %fp = ffbef680, %sp = ffbef620
# fact (1): %fp = ffbef620, %sp = ffbef5c0
```

```
n! = 720
```

(d) Dans un processeur où la multiplication n'est pas implémentée par un circuit, celle-ci peut être réalisée efficacement en se basant sur l'algorithme suivant :

$$
\begin{aligned}
a \times b := & \ \textbf{si } b = 0 \textbf{ alors } 0 \\
& \ \textbf{sinon si } b = 2 \times b' \textbf{ alors } (a \times 2) \times b' \\
& \ \textbf{sinon } ((a \times 2) \times b') + a
\end{aligned}
$$

En vous appuyant sur cette méthode, proposez une fonction assembleur réalisant la multiplication entière en utilisant uniquement des additions et des décalages.

Correction :

Deux versions,

– *une version récursive :*

```
!!!
!!! fonction multiplication: version récursive
!!!

.section        ".text"
        .align 4
        .global mult

mult:
        save %sp, -96, %sp

        cmp 0, %i1              ! if (%i1 != 0) {
        be zero                 !
        nop                     !
        sll %i0, 1, %o0         !     %o0 <- %i0 << 1
        srl %i1, 1, %o1         !     %o1 <- %i1 >> 1
        call mult               !     %o0 <- mult (%o0, %o1)
        nop                     !
        andcc %i1, 1, %g0       !     if (%i1 & 1 != 0) {
        bz even                 ! // -> équivalent à
        nop                     ! //  if (%i1 % 2 == 1) {
        add %o0, %i0, %o0       !         %o0 <- %o0 + %i0
even:                           !     }
        mov %o0, %i0            !     %i0 <- %o0
        b return                !
        nop                     !
zero:                           ! } else {
        mov 0, %i0              !     %i0 <- 0
                                ! }
return:
        ret                     ! return %i0
        restore
```

– *une version itérative :*

```
!!!
!!! fonction multiplication: version itérative
!!!

.section        ".text"
        .align 4
        .global mult

mult:
```

```
        save %sp, -96, %sp

        clr %l0                 ! %l0 <- 0
loop:
        cmp 0, %i1              ! while (%i1 != 0) {
        be end_loop             !
        nop                     !
        andcc %i1, 1, %g0       !     if (%i1 & 1 != 0) {
        bz even                 ! // -> équivalent à
        nop                     ! //  if (%i1 % 2 == 1) {
        add %l0, %i0, %l0       !         %l0 <- %l0 + %i0
even:                           !
        sll %i0, 1, %i0         !     %i0 <- %i0 << 1
        srl %i1, 1, %i1         !     %i1 <- %i1 >> 1
        b loop                  ! }
        nop
end_loop:
        mov %l0, %i0            ! return %l0
        ret
        restore
```

(e) Utilisez la fonction précédente dans un programme C. L'exécutable s'obtient en donnant simplement le fichier C et le fichier assembleur au programme `gcc`, ce dernier s'occupe de faire les liens.

Correction :

```
#include <stdio.h>

extern unsigned mult (unsigned, unsigned);

int main (void)
{
    unsigned a, b;

    printf ("a, b? ");
    scanf ("%u %u", &a, &b);
    printf ("  %u x %u = %u\n", a, b, mult (a, b));

    return 0;
}
```

(f) Écrivez un programme assembleur de recherche de l'élément minimum d'un tableau. Le tableau est une variable locale à la routine principale. Cette dernière fait appel à une routine pour la recherche de l'élément minimum d'un tableau.

Correction :

```
!!!
!!! recherche du minimum dans un tableau
!!!

! Fonction min: recherche du minimum dans un tableau d'entiers
!       2 arguments: adresse du tableau et nombre d'éléments
!       retourne l'indice du minimum dans le tableau
!       précondition: la tableau contient au moins 1 élément
!------------------------------------------------------------

.section        ".text"                         ! -> code
        .align 4
        .global min
```

```
min:
        save %sp, -64, %sp
                                                ! registres locaux utilisés:
                                                ! %l0: index du min. courant
                                                ! %l1: minimum courant
                                                ! %l2: index courant
                                                ! %l3: valeur courante

        clr %l0                                 ! %l0 <- 0
        ld [%i0], %l1                           ! %l1 <- tab[0]
        clr %l2                                 ! %l2 <- 0
        orcc %i1, %g0, %g0
min_loop:                                       ! while (%i1 != 0) {
        bz end_min                              !
        nop                                     !
        inc %l2                                 !     %l2 ++
        add %i0, 4, %i0                         !     %i0 <- %i0
                                                !            + sizeof(int)
        ld [%i0], %l3                           !     %l3 <- tab[%l2]
        cmp %l1, %l3                            !     if (%l1 > %l3) {
        ble min_ok                              !
        nop                                     !
        mov %l2, %l0                            !         %l0 <- %l2
        mov %l3, %l1                            !         %l1 <- %l3
min_ok:                                         !     }
        deccc %i1                               !     %i1 --
        b min_loop                              ! }
        nop

end_min:
        mov %l0, %i0                            ! return %l0
        ret
        restore


! Programme principal
!       -> lit 10 entiers et trouve le minimum
!------------------------------------------------------------

.section        ".data"                         ! -> données
        .align 8
.PRINTF1:
        .asciz "Entrez 10 entiers:\n"
.SCANF:
        .asciz "%d"
.PRINTF2:
        .asciz "minimum: [%d] = %d\n"


.section        ".text"                         ! -> code
        .align 4
        .global main

main:
        save %sp, -136, %sp                     ! réserve de la place pour un
                                                ! tableau de 10 entiers à
                                                ! l'adresse %fp-40
        set .PRINTF1, %o0
```

```
        call printf                     ! printf (.PRINTF1)
        nop

                                        ! lecture des entiers
        mov 10, %l0                     ! %l0 <- 10 (nombre d'entiers
                                        !           à lire)
        sub %fp, 40, %l1                ! %l1 <- %fp-40 (adresse du
                                        !               tableau)
read_loop:                              ! do {
        set .SCANF, %o0                 !
        mov %l1, %o1                    !
        call scanf                      !     scanf (.SCANF, %l1)
        nop                             !
        add %l1, 4, %l1                 !     %l1 <- %l1
                                        !           + sizeof (int)
        deccc %l0                       !     %l0 --
        bnz read_loop                   ! } while (%l0 != 0)
        nop

                                        ! calcul du minimum
        sub %fp, 40, %o0                ! %o0 <- %fp-40 (adresse du
                                        !               tableau)
        mov 10, %o1                     ! %o1 <- 10 (taille du
                                        !           tableau)
        call min                        ! %o0 <- min (%o0, %o1)
        nop

                                        ! affichage du résultat
        mov %o0, %o1                    ! %o1 <- %o0 (indice du min.)
        sll %o0, 2, %o0                 ! %o0 <- %o0 * 4
                                        !       (4 == sizeof (int))
        sub %o0, 40, %o0                ! %o0 <- %o0 - 40
        ld [%fp+%o0], %o2               ! %o2 <- [%fp+%o0] (tab[%o1],
                                        !            valeur du min.)
        set .PRINTF2, %o0
        call printf                     ! printf (.PRINTF2,
                                        !        indice_du_min,
                                        !        valeur_du_min)
        nop

        clr %i0                         ! return 0
        ret
        restore
```

5 TP 2 : DLXview

Analyse d'un programme dlx

Récupérer le programme *example.s*. Celui-ci doit être lancé après avoir initialisé le registre r1 à l'adresse 0 : soit en tapant préalablement dans le terminal de contrôle **put r1 0** soit en chargeant un fichier d'initialisation *example.i*. Il faut, de même, initialiser r4 à zéro.
Url du programme : `http://icps.u-strasbg.fr/~vivien/Enseignement/Archi-2001-2002/example.i`

Questions :

– Sans l'exécuter, dites ce que fait ce programme.

Correction : *Ce programme contient un tableau d'entiers placé à partir de l'adresse 0 (soit* tab *ce tableau).*

Pendant l'exécution, il lit tab[0] *entiers à partir de* tab[1]*, les somme et enregistre le résultat dans* tab[tab[0] + 1] *(à la fin du tableau).*

– Chargez ensuite le programme dans dlxview. Dans le terminal de contrôle, affichez le contenu de la mémoire pour vérifier que celle-ci contient bien les données ainsi que les instructions.

Correction : *On a défini 4 entiers à partir de l'adresse 0 :*

```
(dlxview) get 0 4d
0x0:    3
0x4:    9
0x8:    -14
0xc:    11
```

Le code est placé à l'adresse 256, il y a 9 instructions :

```
(dlxview) get 256 9i
_main:  lw r2,0x0(r1)
loop:   addi r1,r1,0x4
loop+0x4:       lw r3,0x0(r1)
loop+0x8:       add r4,r4,r3
loop+0xc:       subi r2,r2,0x1
loop+0x10:      bnez r2,loop
loop+0x14:      addi r1,r1,0x4
loop+0x18:      sw 0x0(r1),r4
loop+0x1c:      trap 0x0
```

– Exécutez ce programme avec dlxview. On s'aperçoit qu'il ne produit pas le résultat escompté. Cherchez la cause de cet échec en réalisant une exécution pas à pas et en consultant régulièrement le contenu des registres. Que signifient les « `stall` » qui apparaissent dans le pipeline ? Modifiez ensuite le programme pour rendre son exécution correcte.

Correction : *On remarque que l'instruction* `addi r1,r1,0x4` *à l'adresse* `loop+0x14` *est exécutée avant de faire le* `bnez r2,loop`*. Il y a donc un* delay slot *pour les branchements (comme en assembleur* SPARC*).*
Les « `stall` » signifient que l'exécution d'une instruction est retardée jusqu'à ce que la précédente soit complètement exécutée. On peut observer que cela se produit quand une instruction suit un chargement en mémoire et qu'elle utilise le même registre, où quand un branchement conditionnel utilise le résultat de l'instruction le précédent.
On corrige le programme en ajoutant un `nop` *après l'instruction* `bnez` *:*

```
(dlxview) get 256 9i
_main:  lw r2,0x0(r1)
loop:   addi r1,r1,0x4
loop+0x4:       lw r3,0x0(r1)
loop+0x8:       add r4,r4,r3
loop+0xc:       subi r2,r2,0x1
loop+0x10:      bnez r2,loop
loop+0x14:      nop
loop+0x18:      addi r1,r1,0x4
loop+0x1c:      sw 0x0(r1),r4
loop+0x20:      trap 0x0
```

Exercice de programmation

L'objectif est de réaliser un programme qui trouve le plus petit et le plus grand entier d'un tableau. Dans le programme, ce tableau sera donné dans la zone `data` qui contiendra le nombre d'éléments du tableau puis les éléments. Les deux résultats doivent être inscrits dans la zone mémoire à la suite des éléments du tableau.

Correction : *Une solution s'exécutant en 93 cycles est :*

```
;       Données placées en mémoire à l'adresse 0
; .word introduit des entiers signés, pour les autres types, on
; utilise .float, .double, .byte

.data 0
.word 6, 1, 9, 6, -14, 11, 2
.space 4
.space 4

;       Instructions placées en mémoire à l'adresse 256 (par défaut)

.text
_main:  xor r1, r1, r1          ; r1: pointeur <- 0
        lw r2, 0(r1)            ; r2: compteur <- *r1
        addi r1, r1, 4          ; r1 ++
        lw r3, 0(r1)            ; r3: min <- *r1
        lw r4, 0(r1)            ; r4: max <- *r1

loop:
        subi r2, r2, 1          ; while (-- r2 != 0) {
        beqz r2, end_loop
        nop

        addi r1, r1, 4          ;       r1 ++
        lw r5, 0(r1)            ;       r5 <- *r1
        slt r6, r5, r3          ;       if (r5 < min) {
        beqz r6, notmin
        nop
        ori r3, r5, 0           ;               min <- r5
        j loop                  ;               continue
        nop
notmin:
        sgt r6, r5, r4          ;       } else if (! (r5 > max)) {
        beqz r6, loop           ;               continue
        nop                     ;       } else {
        ori r4, r5, 0           ;               max <- r5
        j loop                  ;               continue
        nop                     ;       }
                                ; }
end_loop:
        addi r1, r1, 4          ; r1 ++
        sw 0(r1), r3            ; *r1 <- min
        addi r1, r1, 4          ; r1 ++
        sw 0(r1), r4            ; *r1 <- max
        trap 0

; trap 0 provoque la sortie du programme, sans cela, le programme
;       exécuterait à l'infini des instructions nop
```

Dans un deuxième temps, vous ordonnerez au mieux vos instructions pour minimiser le nombre de `stall`.

Correction : *En éliminant tous les* `stall` *et en essayant de minimiser le nombre de* `nop`*, on obtient une solution s'exécutant en 65 cycles :*

```
;       Données placées en mémoire à l'adresse 0
; .word introduit des entiers signés, pour les autres types, on utilise
; .float, .double, .byte

.data 0
.word 6, 1, 9, 6, -14, 11, 2
```

```
.space 4
.space 4

;       Instructions placées en mémoire à l'adresse 256 (par défaut)

.text
_main:  xor r1, r1, r1          ; r1: pointeur <- 0
        lw r2, 0(r1)            ; r2: compteur <- *r1
        addi r1, r1, 4          ; r1 ++
        subi r2, r2, 1          ; r2 --
        lw r3, 0(r1)            ; r3: min <- *r1
        lw r4, 0(r1)            ; r4: max <- *r1

loop:
        beqz r2, end_loop       ; while (r2 != 0)
        addi r1, r1, 4          ;       r1 ++ (delay slot)

        lw r5, 0(r1)            ;       r5 <- *r1
        subi r2, r2, 1          ;       r2 --
        slt r6, r5, r3          ;       r6 <- (r5 < min)
        sgt r7, r5, r4          ;       r7 <- (r5 > max)
        beqz r6, notmin         ;       if (r6) {
        nop
        j loop                  ;               continue
        ori r3, r5, 0           ;               min <- r5 (delay slot)
notmin:                         ;       } else if (! r7) {
        beqz r7, loop           ;               continue
        nop                     ;       } else {
        j loop                  ;               continue
        ori r4, r5, 0           ;               max <- val (delay slot)
                                ;       }
                                ; }
end_loop:                       ; }
        sw 0(r1), r3            ; *r1 <- min
        addi r1, r1, 4          ; r1 ++
        sw 0(r1), r4            ; *r1 <- max
        trap 0

; trap 0 provoque la sortie du programme, sans cela, le programme
; exécuterait à l'infini des instructions nop
```

6 Projet assembleur SPARC

6.1 Fonction modulo

Écrire une fonction calculant le reste de la division entière (fonction *modulo*) de son premier argument par le second. Les arguments sont des entiers positifs, le deuxième est non nul. Vous pourrez utiliser la formule suivante :

$$a \bmod b = a - b \times \left\lfloor \frac{a}{b} \right\rfloor$$

où $\lfloor \frac{a}{b} \rfloor$ est le résultat de la division entière de a par b.

Correction :

```
.section        ".text"
        .align 4
        .global modulo
```

```
modulo:                         ! modulo (a, b): reste (r) de la division
                                !               entière a / b
                                !       a = b.q + r
                                !  <=>  r = a - b.q
                                ! avec  q = a / b
        save %sp, -64, %sp

        mov %g0, %y             ! le premier opérande de udiv est sur 64 bits
        udiv %i0, %i1, %i2      ! %i2 <- q = a / b
        umul %i1, %i2, %i1      ! %i1 <- b.q
        sub %i0, %i1, %i0       ! %i0 <- r = a - b.q

        ret                     ! return %i0
        restore
```

6.2 Algorithme d'Euclide

6.2.1 Version itérative

Écrire un programme prenant en entrée deux nombres entiers positifs *a* et *b*, et calculant puis affichant le pgcd de ces deux nombres. Le pgcd sera calculé avec une version itérative de l'algorithme d'Euclide :

tant que b $\neq$ 0
  r := a mod b
  a := b
  b := r
fin tant que
pgcd := a

Vous utiliserez la fonction *modulo* écrite à la question 6.1.

Correction :

```
.section        ".rodata"
        .align 8

.INPUT_PRINTF:
        .asciz "a b (a >= b)? "

.INPUT_SCANF:
        .asciz "%u %u"

.OUTPUT_PRINTF:
        .asciz "pgcd(%u, %u) = %u\n"


.section        ".text"
        .align 4
        .global main

main:
        save %sp, -104, %sp     ! réserve de l'espace pour 2 variables locales

        set .INPUT_PRINTF, %o0
        call printf             ! printf (.INPUT_PRINTF)
        nop

        set .INPUT_SCANF, %o0
        add %fp, -4, %o1
```

```
        add %fp, -8, %o2
        call scanf              ! scanf (.INPUT_SCANF, %fp-4, ,%fp-8)
        nop

        ld [%fp-4], %l0         ! %l0 <- *(%fp-4)
        ld [%fp-8], %l1         ! %l1 <- *(%fp-8)

.while:                         ! # calcul de %l0 mod %l1
        cmp %l1, %g0            ! while (%l1 != 0) {
        be .end_while           !
        nop                     !

        mov %l0, %o0            !
        mov %l1, %o1            !
        call modulo             !    %o0 <- %l0 mod %l1
        nop                     !

        mov %l1, %l0            !    %l0 <- %l1
        mov %o0, %l1            !    %l1 <- %o0

        ba .while               ! }
        nop

.end_while:                     ! # le résultat est dans %l0

        set .OUTPUT_PRINTF, %o0
        ld [%fp-4], %o1
        ld [%fp-8], %o2
        mov %l0, %o3
        call printf             ! printf (.OUTPUT_PRINTF,
                                !         *(%fp-4), *(%fp-8), %l0)
        nop

        clr %i0
        ret                     ! return 0
        restore
```

6.2.2 Version récursive

Même question qu'en 6.2.1, mais avec une version récursive de l'algorithme :

pgcd(a, b) := **si** b = 0 **alors** a **sinon** pgcd(b, a mod b)

Correction :

```
.section        ".text"
        .align 4
        .global pgcd

pgcd:                           ! pgcd (%i0, %i1):  calcul du pgcd
        save %sp, -96, %sp

        cmp %i1, %g0            ! if (b != 0) {
        be .end_pgcd            !
        nop                     !

        mov %i0, %o0            !
        mov %i1, %o1            !
        call modulo             !       %o0 <- %i0 mod %i1
```

```
        nop                     !

        mov %o0, %o1            !
        mov %i1, %o0            !
        call pgcd               !       %o0 <- pgcd (%i1, %o0)
        nop                     !

        mov %o0, %i0            !       %i0 <- %o0
.end_pgcd:                      ! }
        ret                     ! return %i0
        restore


.section        ".rodata"
        .align 8

.INPUT_PRINTF:
        .asciz "a b (a >= b)? "

.INPUT_SCANF:
        .asciz "%u %u"

.OUTPUT_PRINTF:
        .asciz "pgcd(%u, %u) = %u\n"


.section        ".text"
        .align 4
        .global main

main:
        save %sp, -104, %sp     ! réserve de l'espace pour 2 variables locales

        set .INPUT_PRINTF, %o0
        call printf             ! printf (.INPUT_PRINTF)
        nop

        set .INPUT_SCANF, %o0
        add %fp, -4, %o1
        add %fp, -8, %o2
        call scanf              ! scanf (.INPUT_SCANF, %fp-4, ,%fp-8)
        nop

        ld [%fp-4], %o0
        ld [%fp-8], %o1
        call pgcd               ! %o0 <- pgcd (*(%fp-4), *(%fp-8))
        nop

        mov %o0, %o3
        set .OUTPUT_PRINTF, %o0
        ld [%fp-4], %o1
        ld [%fp-8], %o2
        call printf             ! printf (.OUTPUT_PRINTF, *(%fp-4), *(%fp-8), %o0)
        nop

        clr %i0
        ret                     ! return 0
        restore
```